THE UNIVERSITY OF MELBOURNE

COMP 90048 Declarative Programming Workshop 3 (week4)

2019 semester 1

by Wendy Zeng

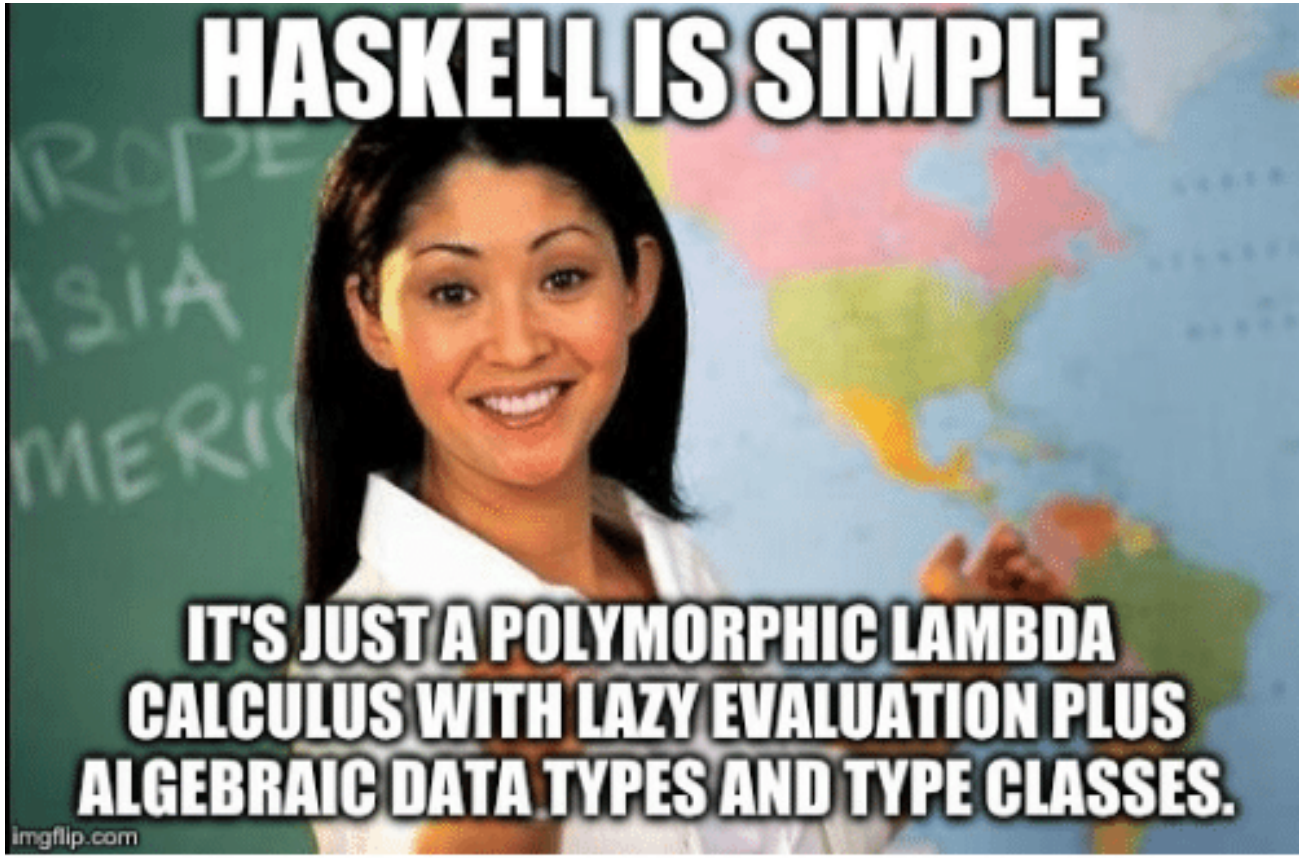
Tutorial : Tue 18:15 - 19:15 221 Bouverie St, room B11

Wed 17:15 - 18:15 201 Bouverie St, room B132

Outline

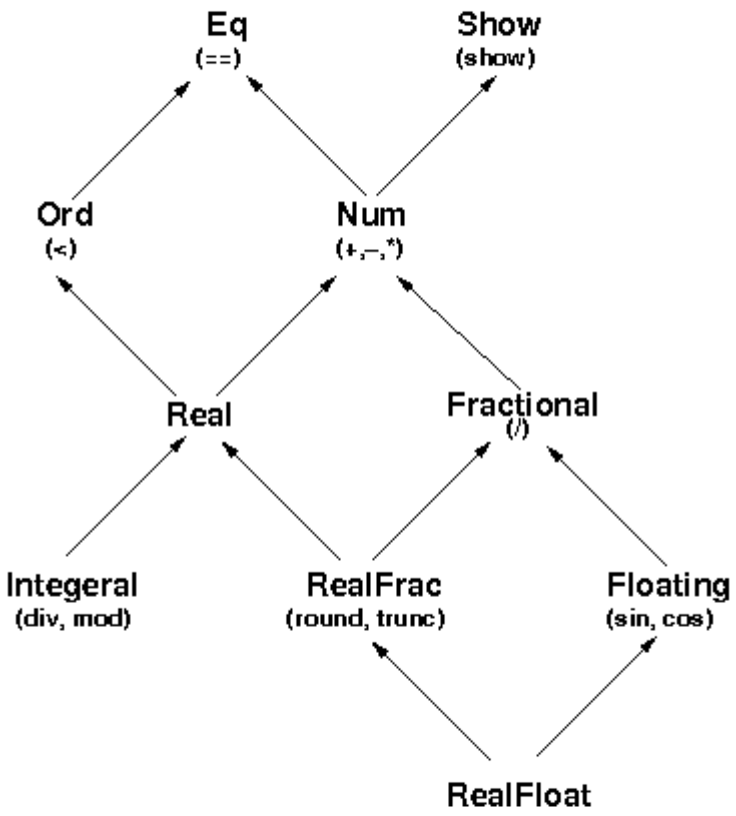
1. Brief overview of Haskell type class and type system
2. Recursive data structure and pattern matching
3. Implementing quick sort and merge sort

1. Brief overview of Haskell type class and type system



Haskell is an easy to learn language

1. Brief overview of Haskell type class and type system



(http://academic.udayton.edu/saverioperugini/courses/cps343/lecture_notes/Haskell.html)

- **Type class:**
  - Define **a set of methods** applicable for all types that derive this specific class
  - Inheritance
- **Type:**
  - Instances of certain type class
  - **Derive** certain type classes to be eligible to apply certain methods

1. Brief overview of Haskell type class and type system

- **Eq and Ord:**
  - Eq: ==, /=
  - Ord: <, >, <=, >=, max, min

- **Read and Show**

- **Num:**
  - **Fractional**: /, recip, fromRational
  - **Floating**: pi, exp, log, sqrt, cin, cos

1. Brief overview of Haskell type class and type system

Which one of the following is the most concise, correct type for a function that determines whether all elements of a list are in strictly increasing order?

- (Eq a, Ord a) => [a] -> Bool
- (Ord a, Eq b) => [a] -> b
- [a] -> Bool
- Ord t => [t] -> Bool
- Ord t => [t] -> Int

Using Ord implies its eligibility for Eq (Ord ‘inherits’ Eq)

2. Recursive data structure and pattern matching

- Type of data structures:
  - Non-recursive data structure

```
data Card = Card Suit Rank
```

  - Self-recursive data structure: **arguments of data constructor are of type of itself**

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
```

```
data Expr = Number Int
          | Variable String
          | Binop Binop Expr Expr
          | Unop Unop Expr
data Binop = Plus | Minus | Times | Divide
data Unop = Negate
```

2. Recursive data structure and pattern matching

- Mutually-recursive data structure: **at least one of them must have a non-recursive alternative**

```
data BoolExpr = BoolConst Bool
              | BoolOp BoolOp BoolExpr BoolExpr
              | CompOp CompOp IntExpr IntExpr
data IntExpr = IntConst Int
             | IntOp IntOp IntExpr IntExpr
             | IntIfThenElse BoolExpr IntExpr IntExpr
```

2. Recursive data structure and pattern matching

```
maybeAdd :: Maybe Integer -> Maybe Integer -> Maybe Integer
maybeAdd (Just x) (Just y) = Just (x+y)
maybeAdd _ _ = Nothing
```

Should not be limited to only Integer type

```
maybeAdd :: Num a => Maybe a -> Maybe a -> a
maybeAdd (Just x) (Just y) = x+y
maybeAdd _ _ = 0
```

Unnecessary to unwarp the value within Monad class Maybe (base case not correct either)

```
maybeAdd :: Num a => Maybe a -> Maybe a -> Maybe a
maybeAdd (Just x) (Just y) = Just (x+y)
```

Non-exhaustive pattern matching

```
maybeAdd :: Num a => Maybe a -> Maybe a -> Maybe a
maybeAdd (Just x) (Just y) = Just (x+y)
maybeAdd _ _ = Nothing
```

Most concise implementation (will see other options later in the semester using Monad)

3. Implementing quick sort and merge sort

- **Quicksort:**

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (pivot:xs) = quicksort smaller ++ [pivot] ++ quicksort larger
        where
        smaller = [ x | x <- xs, x<pivot ]
        larger   = [ x | x <- xs, x>=pivot  ]

… or
        where
        smaller = filter (< pivot)  xs
           larger   = filter (>= pivot) xs
```

3. Implementing quick sort and merge sort

- **Merge sort:**

  see workshop11 Q3

Thank you

wendy.zeng@unimelb.edu.au

By Wendy Zeng